Centro de Investigación en Matemáticas A.C.

Maestría en Cómputo Estadístico

Implementación de Algoritmo Genético y Memético para solución del TSP mediante un entorno distribuido

PRESENTA

Yair Antonio Castillo Castillo
José Antonio Duarte Mendieta

ASIGNATURA

Tópicos Selectos en Aprendizaje Máquina y BigData

9 de diciembre de 2021

Índice general

Capítulo 1

Introducción

Los problemas de recorridos han sido ampliamente estudiados en las últimos años. Buscar maneras más eficientes de realizar la distribución de mercancías es una prioridad desde el ámbito logístico, reducir los costos es ampliar la competitividad, en la distribución logística son muchos los aspectos que pueden ser mejorados. En el transcurso de todos estos años se han formulado una amplia variedad de problemas de ruteo que tienen su estructuración más simple a partir del TSP el cual es un problema combinatorio fácil de formular que pertenece a la clase NP-completos, lo que significa que es difícil de resolver.

El problema del TSP (Travelling Salesman Problem) o el problema del viajero, se puede ver como: "Un vendedor debe visitar una vez y solo una vez cada una de las n ciudades diferentes comenzando desde una ciudad base y regresando a esta ciudad. ¿Qué camino minimiza la distancia total recorrida por el vendedor?"

Los algoritmos genéticos y meméticos son técnicas evolutivas que utilizan operadores de cruce y mutación para resolver problemas de optimización difíciles y optimización combinatoria utilizando la idea de supervivencia del más apto. Se han utilizado con éxito en una variedad de problemas diferentes, incluido el TSP.

También se ha visto que el hecho de paralelizar un problema hace que el tiempo de computo disminuya en una gran cantidad. Por lo que para acelerar la búsqueda de la solución óptima se usará el framework Spark, en ese caso se usa su implementación en python, es decir, PySpark.

Capítulo 2

Objetivo del Proyecto

Por lo que el objetivo de este proyecto es hacer un modelo para resolver el problema de TSP (Travelling Salesman Problem) o el problema del viajero utilizando algoritmos genéticos y meméticos en un entorno distribuido como sería PySpark. Para los resultados se grafica los tiempos computacionales haciendo de manera paralela y de manera secuencial. También se muestra la cantidad de población inicial, el número de ciudades y las especificaciones de la computadora donde se corrió estos algoritmos.

Capítulo 3

Metodología

3.1. Problema TSP

3.1.1. Explicación del problema y representación de la solución

El problema que se trata en este proyecto es el Travelling Salesman Problem o TSP, que consiste en encontrar la ruta más corta que un vendedor debería tomar para recorrer un conjunto C de ciudades.

La representación del cromosoma es una representación de orden de la forma

$$[1, 2, ..., n]$$

donde se guardan los índices de las ciudades en el orden en el que se visitarán. Debido a la forma en la que se genera la población inicial y los operadores de muta y de cruza utilizados, cada ciudad aparece sólo una vez en cada cromosoma, y el último viaje se realiza del último elemento en la lista al primero.

La función objetivo calcula la distancia recorrida para visitar las C ciudades según el orden que especifica un cromosoma y puede ser expresada como:

$$z = \sum_{i \in I} \sum_{j \in J} d_{ij} x_{ij}$$

con d_{ij} la distancia de ir de la ciudad i a la ciudad j para toda $i, j \in C \quad i \neq j$ y x_{ij} una variable indicadora que toma el valor uno si en la ruta se va de la ciudad i a la ciudad j y el valor cero en otro caso, y el objetivo de este problema es encontrar el cromosoma que minimiza el valor de la función objetivo.

Como se mencionó anteriormente, por el operador de cruza y el operador de muta (2-opt) que utilizamos y por la representación que se eligió para los cromosomas, nos aseguramos de que se cumplen las restricciones del problema y de que no se producen soluciones infactibles. A continuación se explica como garantizamos que se respeten las restricciones del problema y evitamos soluciones infactibles en el algoritmo:

1) De la ciudad i solo se puede ir a una sola ciudad. Esta restricción se puede representar como

$$\sum_{j \in J} x_{ij} = 1$$

para toda $i \in I$ y dicha restricción se cumple en nuestro algoritmo gracias a la representación de orden, pues para una entrada particular i del cromosoma sólo existe una y sólo una entrada posterior a la entrada i, y dicha entrada sería la ciudad a la que se iría partiendo de la ciudad i.

2) A la ciudad j solo se puede llegar desde una sola ciudad. Esta restricción se representa formalmente como:

$$\sum_{i \in I} x_{ij} = 1$$

para toda $j \in J$. Esta restricción se cumple en nuestro algoritmo debido a la representación de orden, pues para una entrada particular j del cromosoma sólo existe una y solo una entrada anterior a la entrada j, y dicha entrada sería la ciudad desde la que se partió para llegar a la ciudad j.

3) No pueden haber "subtour", es decir, no puede haber recorridos sin pasar por todas las ciudades. Esta restricción se refiere a que las ciudades solo pueden ser visitadas una sola vez en la ruta propuesta como solución. Dicha restricción se puede representar como :

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1$$

para toda $S \subset \{1, 2, 3, ..., n\}$. Esta restricción se cumple en nuestro algoritmo debido a que cuando creamos nuevos cromosomas (i.e. soluciones potenciales), lo hacemos mediante una permutación del conjunto $\{1, 2, 3, ..., n\}$, donde n representa el número de ciudades que pueden ser visitadas. De esta forma, garantizamos que ninguna ciudad se repita más de una vez en un gen, y por lo tanto ninguna ciudad es visitada más de una vez.

4) La variable indicadora x_{ij} solo debe tomar el valor 0 o el valor 1. Esta restricción se representa como

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in C$$

Esta restricción nos permite saber cuando desde una ciudad particular i se viajó a una ciudad particular j. En nuestro algoritmo esta restricción se cumple implícitamente en la representación de orden, pues el orden en el que están ordenados los genes nos permite saber de que ciudad en particular se viaja a otra.

3.2. Generación de la población inicial

La población inicial se genera realizando N permutaciones al azar del conjunto $\{1, 2, 3, ..., n\}$. Por la forma en la que generamos la población inicial y mediante la revisión de la solución en cada iteración, garantizamos que no haya cromosomas repetidos en la población.

Es posible aprovechar la arquitectura distribuida para generar la población inicial de forma más rápida. La tarea de generar cromosomas al azar se paraleliza en tres nodos, donde cada uno de ellos calcula aproximadamente $\frac{N}{3}$ permutaciones distintas. Dentro de cada nodo, se garantiza que las permutaciones sean únicas, sin embargo, es posible que el mismo cromosoma se repita en dos nodos, o incluso en los tres. Para corregir esta situación y contar exactamente con N cromosomas únicos, una vez que los tres nodos terminan de generar cromosomas aleatorios, estos cromosomas se unen y se remueven cromosomas repetidos. Se vuelve a revisar entonces el número de cromosomas en la población resultante, y si se tienen menos de N cromosomas, entonces de forma secuencial se producen y agregan los cromosomas faltantes a la población (revisando que no estén ya en ella, es decir, garantizando que no existan cromosomas repetidos). Una vez que se ha cuenta con N cromosomas en la población, entonces dicha poblacioń se regresa como la población inicial y comienza el proceso de cruza.

A continuación se muestra un esquema de la arquitectura distribuida para este paso:

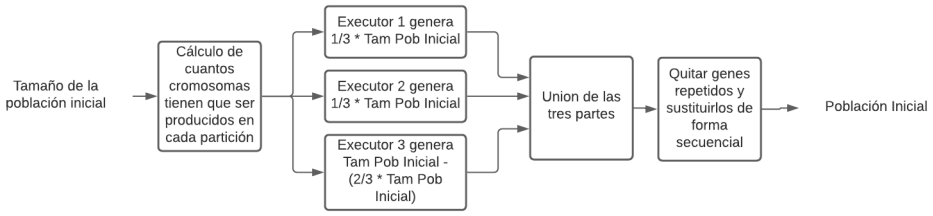


Figura 3.1: Diagrama de creación de población inicial

3.2.1. Pseudocódigo

El pseudocódigo de la generación de población inicial en PySpark

Algorithm 1 Generación de población inicial

Input: *tam_pob_inicial*, N : Tamaño de la población inicial, Número de ciudades
Output: *PoblaciónFinal*: Regresa la población final

ciudades ← Un vector de tamaño N, con los índice de $1, ..., N$
dat_paralel ← Definimos los clusters de datos que se procesarán en los nodos
procedure *generar_parte_aleatoria*(datos)
    parte ← Índice de la partición
    datos_a_gen ← Número de cromosomas a generar
    pob_inicial ← [ ]
    while $len(pob\_inicial) \leq datos\_a\_gen$ **do**
        gen ←permutación de la ciudades
        if not *gen* en *pob_inicial* **then**
            Agregamos *gen* a *pob_inicial*
        end if
    end while
    return *pob_inicial*
end procedure
partes ← Ejecutamos el mapper *generar_parte_aleatoria* en *dat_paralel*
$resultado\_final$ ← Obtenemos los cromosomas únicos de *partes*
Checamos si en $resultado\_final = tam\_pob\_inicial$ sino agregamos diferentes

3.3. Algoritmos Genéticos

Los algoritmos evolutivos son algoritmos de búsqueda probabilística que simulan la evolución natural. Fueron propuestos hace unos 30 años. Su aplicación a los problemas de optimización combinatoria, sin embargo, se convirtió recientemente en un tema de investigación real. En los últimos años se han publicado numerosos artículos y libros sobre la optimización evolutiva de problemas NP-hard, en dominios de aplicación muy diferentes como biología, química, diseño asistido por ordenador, criptoanálisis, identificación de sistemas, medicina, microelectrónica, reconocimiento de patrones, planificación de la producción, robótica, telecomunicaciones, etc.

En los algoritmos genéticos el espacio de búsqueda de un problema se representa como una colección de individuos. Estos individuos están representados por cadenas de caracteres (o matrices), que a menudo se denominan cromosomas. El propósito de utilizar un algoritmo genético es encontrar al individuo del espacio de

búsqueda con el mejor "material genético". La calidad de un individuo se mide con una función de evaluación. La parte del espacio de búsqueda que se examinará se denomina población

Los operadores que definen el proceso de producción de hijos y el proceso de mutación se denominan operador de cruza y operador de muta, respectivamente. La muta y la cruza juegan diferentes roles en el algoritmo genético. La muta es necesaria para explorar nuevos estados y ayuda al algoritmo a evitar los óptimos locales. La cruza debería incrementar la calidad media de la población. Al elegir operadores de muta y de cruza adecuados, aumenta la probabilidad de que el algoritmo genético dé como resultado una solución casi óptima en un número razonable de iteraciones. Puede haber varios criterios para detener el algoritmo genético. Por ejemplo, si es posible determinar previamente el número de iteraciones necesarias como se hace en este caso.

3.3.1. Cruza

Para cada uno de los cromosomas en la población inicial se aplica una probabilidad de cruza (se recomienda un valor entre 0.6 y 0.9) para escoger aquellos cromosomas que entraran a la cruza. Si el padre no entra a la cruza, entonces se toma a él mismo como el hijo y se pasa a la muta.

Para cada cromosoma que haya entrado a la cruza se utiliza el método de emparejamiento por barrera, que consiste en escoger como pareja a un cromosoma elegido al azar (que no sea él mismo) de la población inicial.

Posteriormente, se elige una proporción de los elementos del padre 1 y dichos elementos se pasan al hijo en la misma posición en la que se encontraban en el padre 1. Después se obtienen los elementos del padre 2 que no fueron transmitidos al hijo por el padre 1 y se van colocando en el hijo en el mismo orden en el que aparecen en el padre 2. De esta forma, el hijo heredó genes tanto del padre 1 como del padre 2. Un ejemplo se puede ver en la siguiente imagen

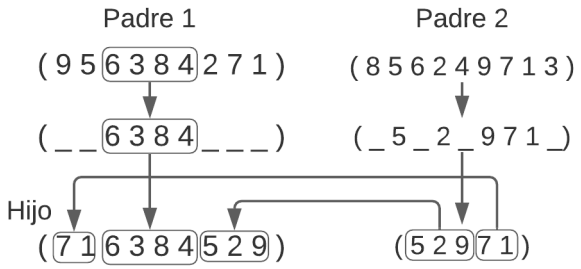


Figura 3.2: Ejemplo de operador de Cruza

3.3.2. Muta

Para la muta el operador que utilizamos lleva el nombre de intercambio (2-opt), que como su nombre lo dice, se escogen 2 genes y se intercambian de lugar, las posiciones de los 2 genes son tomados al azar. Por ejemplo, tenemos una solución dada de la siguiente forma

(9 5 6 3 8 4 2 7 1)

se eligen 2 genes al azar, digamos el 3 y 6, es decir, la ciudad 6 y la ciudad 4 y se intercambian

$$(9\ 5\ \underline{6}\ 3\ 8\ \underline{4}\ 2\ 7\ 1) \Rightarrow (9\ 5\ \underline{4}\ 3\ 8\ \underline{3}\ 2\ 7\ 1)$$

Se recomienda una probabilidad de muta entre 0.1 y 0.4.

3.3.3. Selección de la nueva generación

Para la selección de la nueva generación, se utiliza un modelo estacionario, es decir, a la nueva generación pasan cromosomas tanto de la generación inicial como de los hijos de dicha generación.

Se utiliza un criterio de Elitismo Total para elegir los cromosomas que pasarán a la población final. Se juntan entonces los cromosomas de la población inicial y del pool y se ordenan de acuerdo a su fitness y se eligen los mejores N cromosomas y estos cromosomas son los que pasarán a la población inicial de la siguiente iteración.

3.3.4. Implementación de algoritmo genético

Lo que hacemos es que partir de la población inicial se distribuye a los cluster asignados, se les aplica un mapper con cruza con una probabilidad y luego un mapper de la muta con otra probabilidad. Obtenido el Pool (hijos) se hace la unión con la población inicial y se le aplica un mapper que regresa llave(fitness o dístancia recorrida) - valor(cromosoma o recorrido), luego se le aplica un reducer que lo ordena por llave (de menor a mayor) y solo se agarra el mejor conjunto con el tamaño de la población inicial.

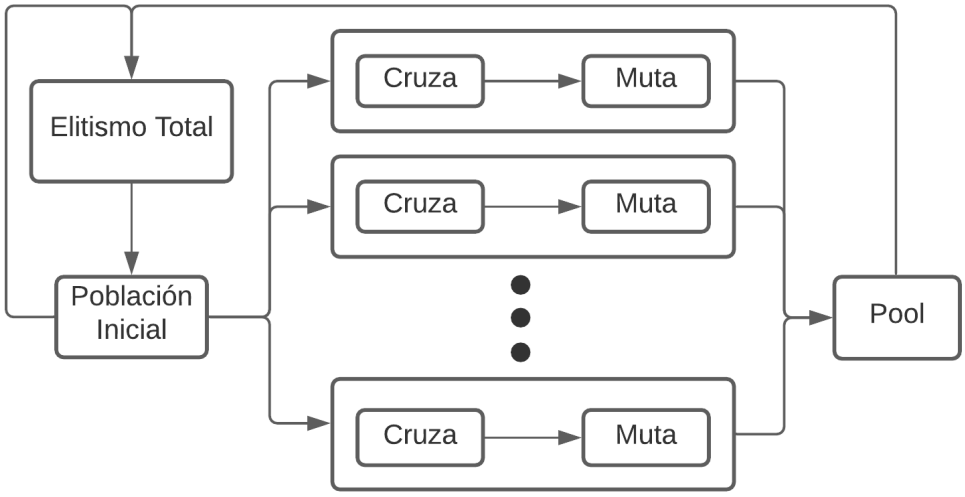


Figura 3.3: Diagrama del algoritmo genético en PySpark

3.3.5. Pseudocódigo

Pseudocódigo para el algoritmo genético en PySpark, primero se definirá el mapper que toma los cromosomas y regresa un par llave-valor que es fitness (distancia recorrida) y el cromosoma (recorrido) (ver Algoritmo 2: Calular el fitness). Luego definimos otra función donde se hará acaba el mapper de la cruza y el mapper de la muta, estos 2 solo regresa el valor que es el cromosoma (ver Algoritmo 3: Cruza y Muta del Algoritmo Genético). Y por último el algoritmo genético que es aplicar para un número de iteraciones (ver Algoritmo 4: Algoritmo Genético)

Algorithm 2 Calcular el fitness (*calc_fitness*)

Input: *gen*, *mat_datos* : Recorrido, Matriz de distancia
Output: *(fitness,gen)*: Una tupla que tiene el su fitness (distancia recorrida) y cromosoma (recorrido)

1: $fitness \leftarrow$ Distancia recorrida por el gen
2: **return** (*fitness*, *gen*): Sería nuestra llave-valor

Algorithm 3 Cruza y Muta del Algoritmo Genético (*algo_genetico*)

Input: *PoblacionRDD*, *prob_cruza*, *prob_muta*, *mat_dist*: La población de cromosomas en RDD, Probabilidad de Cruza, Probabilidad de muta, Matriz de distancia
Output: *PoblacionFinal*: Regresa la población final

procedure CRUZA(*pareja*, *prob_cruza*)
    if *rand()* $\leq$ *prob_cruza* **then**
        hijo $\leftarrow$ Generamos al hijo con *gen_padre1* y *gen_padre2* que están en *pareja*
    else
        hijo $\leftarrow$ *gen_padre*1 está en *pareja*
    end if
    return *hijo*
end procedure
procedure MUTA(*hijo_gen*, *prob_muta*)
    if *rand()* $\leq P_M$ **then**
        Intercambiamos 2 elementos en *hijo*
    end if
    return *hijo*
end procedure
procedure POB(*pareja*)
    return Padre 1
end procedure
hijos $\leftarrow$ Ejecutamos el mapper $CRUZA$ en *PoblacionRDD*
pool $\leftarrow$ Ejecutamos el mapper $MUTA$ en *hijos*
pob_original $\leftarrow$ Ejecutamos el mapper POB en *PoblacionRDD*
pob_total $\leftarrow$ Unimos *pool* y *pob_original* y obtenemos los únicos
pob_total $\leftarrow$ Ejecutamos el mapper *calc_fitness* en *pob_total* usando *mat_dist*, hacemos reduce con llave valor fitness-Recorrido y lo ordenamos
return *PoblacionFinal*: *pob_total* que está ordenada de menor a mayor por fitness

Algorithm 4 Algoritmo Genético (*algoritmo_genetico*)

Input: *Poblacion*,*mat_dist*,*iter prob_cruza*, *prob_muta*,*tam_pob_inicial*: La población de cromosomas, Matriz de distancia,Número de iteraciones, Probabilidad de Cruza, Probabilidad de muta, Tamaño de la población inicial
Output: *pob_inicial*: Regresa la población

for i in 1:iter **do**
    random_pob_inicial $\leftarrow$ Tomamos una muestra de *Poblacion* de tamaño *tam_pob_inicial*
    pob_inicial_par $\leftarrow$ Unimos *Poblacion* y *random_pob_inicial* por columnas
    pob_inicial_rdd $\leftarrow$ Definimos los clusters de datos que se procesarán en los nodos
    pob_inicial $\leftarrow$ aplicamos $algo\_genetico(pob\_inicial\_rdd, prob\_cruza, prob\_muta, mat\_dist)$
    pob_inicial $\leftarrow$ Tomamos los primero *tam_pob_inicial* de *pob_inicial*
end for
return *pob_inicial*: Población después de varias iteraciones

3.4. Algoritmos meméticos

El algoritmo memético (MA) está motivado por el término de Dawkin de un meme, ya que una unidad de información es procesada y mejorada por las partes que se comunican. Se utiliza para abarcar una amplia clase de metaheurísticas. Este método ha demostrado ser de éxito práctico para la solución aproximada de problemas de optimización de NP. Los algoritmos meméticos aprovechan todo el conocimiento disponible sobre el problema y combinan el poder del algoritmo genético y la búsqueda local.

3.4.1. Implementación de algoritmo memético

Para convertir el algoritmo genético presentado anteriormente en un algoritmo memético es necesario incluir una búsqueda local.

Nosotros hacemos esta inclusión mediante el método **2-opt**, que nos pareció el más adecuado dada la representación de orden que se maneja en este problema. Este método consiste en seleccionar dos genes al azar del cromosoma e intercambiarlos (lo que esencialmente es una búsqueda local). Dicho método lo aplicamos únicamente en las mejores soluciones encontradas una vez que ha terminado de ejecutarse el algoritmo genético. Para aprovechar la arquitectura distribuida, aplicamos la búsqueda local a todos los cromosomas de la población. Para cada cromosoma, realizamos varias búsquedas locales (éste es un parámetro). Si se encuentra que la búsqueda local encuentra una mejor solución (con respecto al fitness), entonces se intercambian las soluciones y, en caso contrario (i.e. la búsqueda local no arroja una mejor solución) entonces conservamos la solución que habíamos encontrado anteriormente. El 2-opt garantiza que las soluciones encontradas en la búsqueda local siempre sean factibles.

Una vez finalizado este proceso con todos cromosomas que entraron a la búsqueda local, volvemos a ordenar el mejor conjunto de soluciones de acuerdo a su fitness y elegimos la mejor solución, que es la que regresamos en la función.

La implementación de esta búsqueda se hace de forma distribuida sobre cada cromosoma. A continuación se muestra un diagrama esquemático de la implementación bajo la arquitectura distribuida:

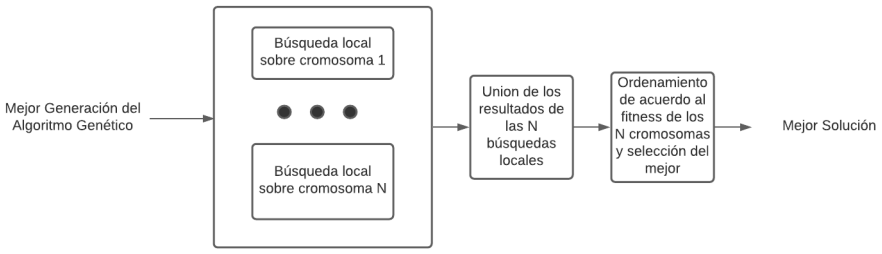


Figura 3.4: Diagrama del Algoritmo Memético en PySpark

3.4.2. Pseudocódigo

El Pseudocódigo para el algoritmo memético

Algorithm 5 Algoritmo Memético (*algo_memetico*)

Input: *PoblacionRDD*, *num_busq_locales*, *matdist*: La población de cromsomas en RDD, Número de iteraciones, Matriz de distancia

Output: *mejor_crom*: Regresa el mejor gen

1: **procedure** BUSQUEDA_LOCAL(*gen*, *num_busq_locales*)
2: *mejor_fitness* ← Obtenemos la distancia (Fitness) de *gen*
3: **for** i in 1:*num_busq_locales* **do**
4: *gen_mutado* ← Intercambio de 2 elementos del *gen*
5: *fitness_gen_mutado* ← Obtenemos la distancia (Fitness) de *gen_mutado*
6: **if** *fitness_gen_mutado* ≤ *mejor_fitness* **then**
7: Intercambiamos *gen* por *gen_mutado*
8: Intercambiamos *mejor_fitness* por *fitness_gen_mutado*
9: **end if**
10: **end for**
11: **return** (*mejor_fitness*,*gen*)
12: **end procedure**
13: *mejor_crom* ← Ejecutamos el mapper *Busqueda_Local* en *PoblacionRDD* usando *num_busq_locales*, hacemos reduce con llave valor fitness-Recorrido y lo ordenamos y tomamos el primer valor
14: **return** *mejor_crom*:Regresa el mejor cromosoma con su fitness

Capítulo 4

Diseño Experimental

Las condiciones bajo las cuáles se llevará a cabo la evaluación del desempeño de nuestra implementación con respecto a los paquetes son las siguientes:

- Python 3.9.6
- Spark 3.1.2
- PySpark 2.1.0

Las evaluaciones se realizan en una computadora con ocho núcleos (lo que determina el nivel de paralelización en spark cuando se trabaja de forma local), 16 gb de RAM y sistema operativo Ubuntu 20.04.

Se realizarán distintas evaluaciones variando los siguientes parámetros:

- Cantidad de cromosomas en la población inicial
- Número de iteraciones en el algoritmo genético
- Número de búsquedas locales en el algoritmo memético

Los valores que se prueban para la población inicial son:

$$\{100,1000,10000,100000,1000000\}$$

Los valores que se prueban para el número de iteraciones en el algoritmo genético son:

$$\{5,10,50,100,1000\}$$

Los valores que se prueban para el número de búsquedas locales en el algoritmo memético son:

$$\{5,10,50,100\}$$

Las evaluaciones se harán por separado, teniendo tres grandes partes para la implementación:

- Generación de la población inicial
- Algoritmo Genético
- Algoritmo Memético

Todas estas evaluaciones se llevarán a cabo en la misma computadora, para garantizar la consistencia. La métrica de interés en este caso es el tiempo de ejecución, medido utilizando la función mágica **time**. Como métrica complementaria se utilizará también el fitness (es decir, la calidad de la solución final), aunque no es

realmente relevante en este proyecto dado que el algoritmo es básicamente el mismo (el cambio principal fue la arquitectura distribuida con la que se implementó), por lo que los resultados son muy similares en cuanto a la calidad de la solución.

También es muy importante mencionar que la paralelización se realiza sobre 8 núcleos, lo cuál se determina en la configuración de Spark.

Para cada evaluación se correrá el algoritmo completo desde el inicio, por lo que se está trabajando con condiciones iniciales particulares para cada paso.

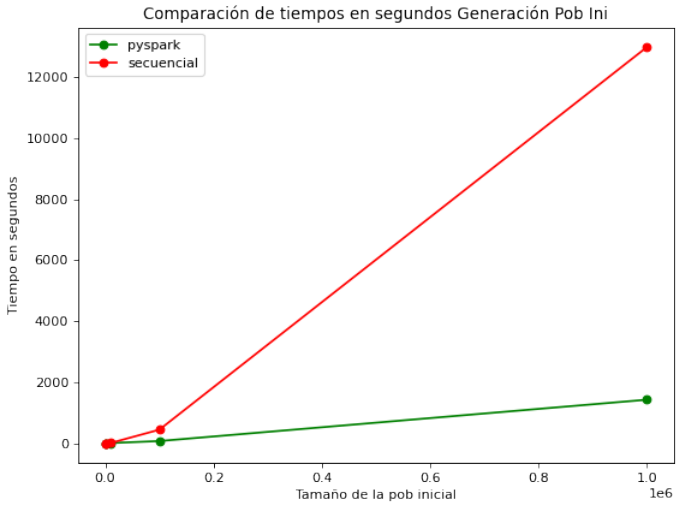
Capítulo 5

Resultados

La primer parte que se evalúa es la generación de la población inicial mediante una arquitectura distribuida contra una arquitectura secuencial. A continuación se presentan los resultados para distintas cantidades de cromosomas que se generarán en la población inicial:

Pob Inicial	Tiempo en Segundos PySpark	Tiempo en Segundos Seq
100	0.009	0.004
1000	0.085	0.077
10000	0.626	2.370
100000	66 .000	438.600
1000000	1422.100	12986.100

La gráfica se muestra a continuación:

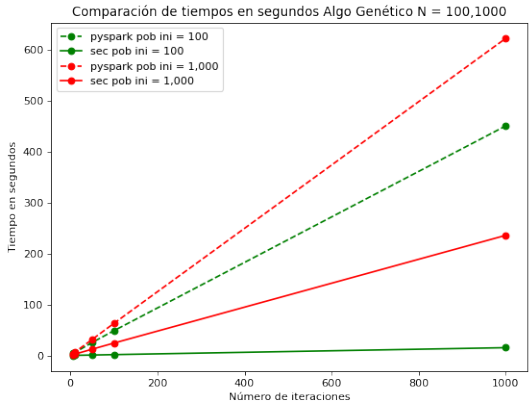


Como es posible observar **el tiempo al utilizar una arquitectura distribuida disminuye significativamente cuando la cantidad de cromosomas a generar también es grande**. **Cuando la cantidad de cromosomas a generar es pequeña, la arquitectura secuencial es más rápida**. Este patrón (cuando se utiliza una gran cantidad de datos es más rápida la arquitectura distribuida y cuando se utiliza una cantidad pequeña es más rápida la arquitectura secuencial) se presenta a lo largo de los experimentos, y puede explicarse considerando que PySpark requiere de la intercomunicación y distribución de datos (lo que implica que hay tiempos extras que se generan cuando es necesario que el driver se comunique con los executors y viceversa). Estos tiempos extras se ven compensados por la paralelización cuando la cantidad de datos es grande, pero cuando los datos son relativamente pocos, estos tiempos extra causan que la arquitectura distribuida sea más lenta que la arquitectura secuencial (la cuál no requiere de ninguna comunicación entre sus partes).

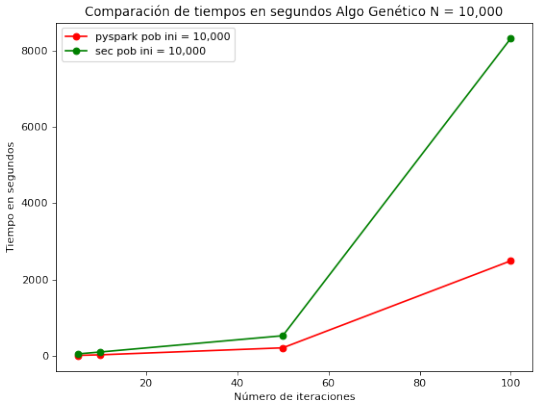
Posteriormente se realiza a cabo la evaluación de la implementación, pero esta vez en la parte correspondiente al algoritmo genético. A continuación se presentan los resultados para distintos valores de la población inicial (N) y para distintos valores para las iteraciones. La siguiente tabla presenta los resultados con respecto al tiempo (en segundos):

Iteraciones	PySpark	Seq	PySpark	Seq	PySpark	Seq	PySpark	Seq
	N = 100		N = 1,000		N = 10,000		N = 100,000	
5	4.03	0.0967	3.16	1.22	14.6	52.6	133	2700
10	5.55	0.165	6.28	2.44	29.6	104	274	NA
50	25.4	0.77	31.4	12.3	213	530	1287	NA
100	48.6	1.58	63	24.3	2492	8320	NA	NA
1000	451	15.3	623	236	NA	NA	NA	NA

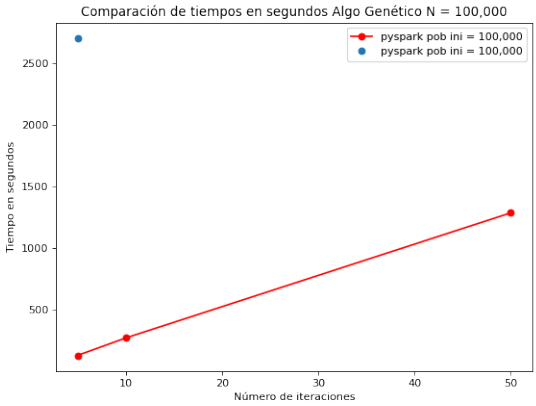
Las gráficas se muestran a continuación para $N = 100$ y $N = 1000$



Para $N = 10,000$:



Y por último para $N = 100,000$:



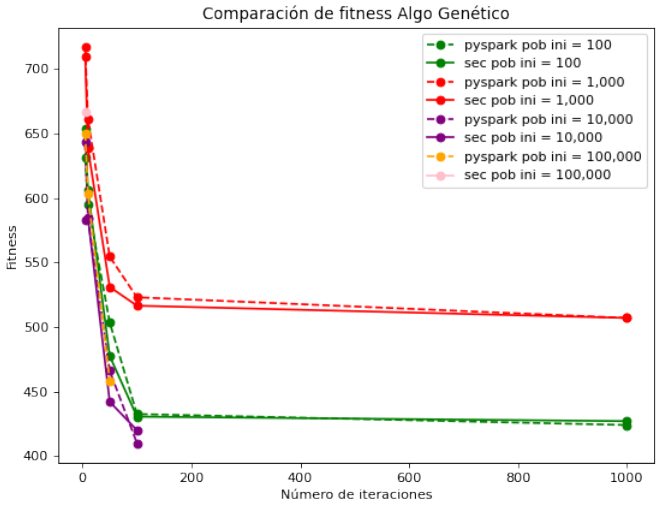
Es posible observar de nuevo que **cuando el número de datos es pequeño ($N = 100$, $N = 1,000$), la arquitectura secuencial es más rápida**, sin embargo, **cuando el número de datos incrementa ($N = 10,000$, $N = 100,000$), la arquitectura distribuida es mucho más rápida** (notar que el intervalo es significativamente más grande con respecto a cuando la arquitectura secuencial es más rápida que la distribuida).

Nota: Los NA implican que no se registró el dato, debido a que los tiempos de cómputo eran muy largos y no fue posible obtenerlos, dadas las restricciones de tiempo en este proyecto.

También se computaron los resultados con respecto al fitness (o calidad de la solución) para los parámetros ya mencionados. En este caso, hay que recordar que como se está trabajando con el problema del TSP, mientras menor sea el fitness mejor es la solución. Los resultados se presentan a continuacion:

Iteraciones	PySpark	Seq	PySpark	Seq	PySpark	Seq	PySpark	Seq
	N = 100		N = 1,000		N = 10,000		N = 100,000	
5	631	654	717	710	643	583	650	667
10	595	606	616.5	638.5	584	585	603	NA
50	503.5	477.5	554.5	531	466	442	458	NA
100	432.5	430.5	523	516.5	432	420	NA	NA
1000	424	427	507	507	NA	NA	NA	NA

La gráfica se muestra a continuación:

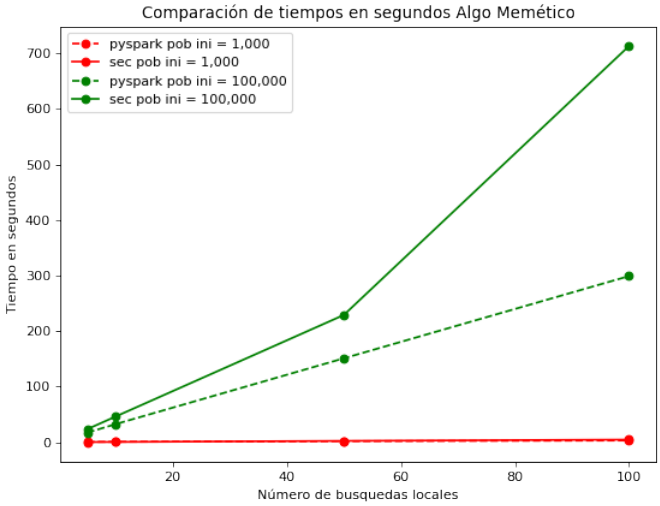


Lo primero que hay que notar, es que **para un N fijo, las diferencias entre las soluciones no son muy distintas**. Esto se debe a que, como se explicó en la metodología, los algoritmos son básicamente los mismos, pues lo único que se modificó fue la arquitectura de éstos. Lo segundo que sí es conveniente notar es que **la calidad de la solución en el algoritmo genético mejora mucho más rápido cuando se aumenta el número de iteraciones que cuando se aumenta el número de cromosomas en la población**. Esto se debe a que los algoritmos genéticos funcionan mediante un proceso de depuración, donde en cada iteración se van obteniendo mejores y mejores soluciones y se van descartando las soluciones no óptimas. También hay que considerar que este proceso de depuración es en esencia secuencial, pues requiere de la población anterior para poder llevar a cabo el proceso de selección, lo que implica que tal vez una arquitectura distribuida no sea la más óptima para esta parte del algoritmo.

Por último, se realizó la evaluación de la implementación en el algoritmo memético. Los resultados con respecto al tiempo cuando se varía el número de búsquedas locales y el número de cromosomas en la población son los siguientes:

Iteraciones	PySpark	Seq	PySpark	Seq
	N = 1,000		N = 100,000	
5	.631	0.262	17.2	23.4
10	.795	0.481	32.6	46.3
50	1.82	2.38	151	229
100	3.09	4.67	299	713

La gráfica se muestra a continuación:

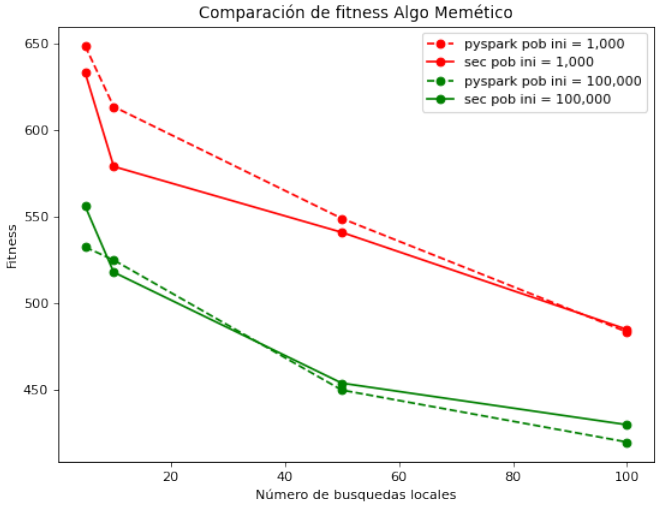


Aquí es posible notar que se tiene de nuevo que **cuando los datos son pocos, la arquitectura secuencial es más rápida, y que cuando aumenta el número de datos, la arquitectura distribuida es más rápida**. También hay que notar que el gap es mucho mayor para cuando la arquitectura distribuida es más rápida.

Los resultados con respecto al fitness (calidad de la solución) son los siguientes:

Iteraciones	PySpark	Seq	PySpark	Seq
	N = 1,000		N = 100,000	
5	648.5	633	532.5	556
10	613.5	579	525	518
50	549	541	450	454
100	483.5	485	420	430

La gráfica se muestra a continuación:



Aquí notamos que **para un valor de fijo de iteraciones, la solución sí mejora significativamente cuando se aumenta el tamaño de la población**, pero también hay una mejora importante (aunque no tan notoria) cuando se aumenta el número de iteraciones (manteniendo fijo el tamaño de la población). Además hay que notar que la naturaleza del algoritmo memético no es secuencial, pues no se requiere de información pasada para llevar a cabo las búsquedas locales (al menos en la implementación del 2-opt). Esto hace que el algoritmo memético funcione muy bien de forma distribuida, en comparación con el algoritmo genético.

Capítulo 6

Conclusiones

Una vez obtenidos los resultados de la evaluación de la implementación, y considerando las características del problema del TSP y de los algoritmos meméticos y genéticos, llegamos a las siguientes conclusiónes:

- La implementación consta de tres grandes partes en donde es posible implementar una arquitectura distribuida: la generación de la población inicial, el algoritmo genético y el algoritmo memético. La conveniencia de dicha arquitectura puede cuantificarse en términos de tiempo y de calidad de la solución.
- Para el caso de la generación de la población inicial, la arquitectura distribuida es una muy buena opción, dado que aunque las tareas que se realizan en esta fase no son completamente independientes entre sí, hacer las generaciones de los cromosomas de forma paralela y posteriormente identificar cromosomas repetidos de forma secuencial mejora significativamente el tiempo necesario en comparación con llevar a cabo toda la tarea de forma secuencial.
- En el caso del algoritmo genético, se notó que las soluciones mejoran de forma significativa en cuanto a su calidad cuando se aumenta el número de iteraciones. Cuando se aumenta el tamaño de la población, las soluciones mejoran, pero no de forma significativa, sin embargo, aumentar el tamaño de la población sí aumenta significativamente los tiempos de ejecución. Considerando esta situación, así como la naturaleza secuencial del algoritmo genético (pues es necesario haber terminado una iteración para poder comenzar con la siguiente), es posible afirmar que esta parte de la implementación no se beneficia mucho de la arquitectura distribuida. Dicho esto, realizar esta parte con arquitectura distribuida cuando se tienen poblaciones muy grandes sí mejora significativamente los tiempos de ejecución.
- En el caso del algoritmo memético, las soluciones mejoran tanto al aumentar el tamaño de la población como el número de búsquedas locales. Las tareas en esta parte del algoritmo son practicamente independientes entre ellas, por lo que se presta a la paralelización y a la arquitectura distribuida. Es posible observar también que cuando se tienen grandes poblaciones, la arquitectura distribuida mejora significativamente los tiempos de ejecución. Esto nos lleva a concluir que el algoritmo memético se beneficia bastante de la implementación hecha aquí.
- Por último, es posible afirmar que cuando se manejan poblaciones pequeñas, la arquitectura secuencial es más rápida que la distribuida (pero es más rápida en cuestión de segundos) mientras que cuando se manejan poblaciones grandes, es mucho más rápida la arquitectura distribuida que la arquitectura secuencial (por cuestión de minutos y a veces incluso por horas). En casos reales, probablemente sea muy provechosa la implementación de una arquitectura distribuida dados los grandes volúmenes de datos que se pueden llegar a utilizar.